Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Розробка та розгортання back-end застосунку на NestJS**»

Виконав: студент 3-го року навчання,

Спеціальності
121 «Інженерія Програмного Забезпечення»

Жорник Дмитро

Керівник Бабич Т.А.

магістр комп'ютерних наук, асистент

«4» червня 2022 р.

Київ – 2021

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2021 року

ЗАВДАННЯ

ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Жорнику Дмитру

1. Тема роботи **«Розробка та розгортання back-end застосунку на NestJS»**, керівник роботи Бабич Трохим Анатолійович, магістр комп’ютерних наук, асистент

2. Строк подання студентом роботи 7 червня 2022

3. План роботи

Анотація

Вступ

Розділ 1. Дослідження та аналіз предметної області

1.1 Дослідження області back-end розробки
1.2 Дослідження інструментів для розгортання back-end застосунків

Розділ 2. Проектування та розробка системи

2.1 Опис складових системи та використаних для розробки технологій

2.2 Налаштування контейнеризації та мережевої взаємодії

2.3 Розробка API

Висновки

Список використаних джерел

Додатки

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2021			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтеня 2021 – 2 листопада 2021			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2021			
4.	Написання розділів роботи	2 листопада 2021 – 01 березня 2022			
5.	Проміжний контроль виконання роботи	01 лютого 2022			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2022 – 29 березня 2022			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2022			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2022			
	Розділ 3 (проектно-рекомендаційна частина)	29 березня 2022			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2022 – 06 травня 2022			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	17 травня 2022			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2021 р.

Науковий керівник Бабич Трохим Анатолійович

Виконавець курсової роботи Жорник Дмитро

ЗМІСТ

Анотація 1

Вступ 2

Розділ 1. Дослідження та аналіз предметної області 4

1.1. Дослідження області back-end розробки 4

1.2. Дослідження інструментів для розгортання back-end застосунків 6

Розділ 2. Проектування та розробка системи 11

2.1. Опис складових системи та використаних для розробки технологій 11

2.2. Налаштування контейнеризації та мережевої взаємодії 12

2.3. Розробка API 15

Висновки 19

Список використаних джерел 19

Додатки 21

Додаток Б 22

Додаток В 23

Додаток Г 25

Додаток Д 26

Додаток Е 27

Додаток Є 28

Анотація

У данній роботі розглядаються особливості розробки та розгортання back-end застосунків мовою TypeScript з використанням фреймворку NestJS. Пояснюється вибір мови TypeScript та фреймворку NestJS та детально описуються етапи розгортання такого застосунку. На прикладі застосунку для ведення складського обліку розглядається прикладний програмний продукт та описуються особливості розробки за допомогою наведених вище технологій.

Вступ

Історично, розробка якісної серверної частини застосунків, або ж так званого back-end була дуже важливою частиною будь-якої інформаційної системи, що працювала в Інтернеті, тому і сформувала до себе надзвичайно високий попит[1]. Також, багатьох інженерів цікавить цей напрям розробки, а отже за весь час розвитку Інтернету було розроблено безліч інструментів для ефективного виконання цієї задачі задля оптимізації ключових показників бізнесу, в тому числі TTM(Time-to-market)[2]. Існує велика кількість різних мов програмування та фреймворків, що дозволяють ефективно розробляти якісні back-end застосунки. Також існує численна множина підходів та засобів для розгортання таких застосунків на серверах, які дозволяють забезпечити високу надійність середовища виконання.

Метою цієї роботи є дослідження технологій, що дозволяють ефективно розробляти та розгортати back-end застосунки; аналіз цих технологій та обґрунтування вибору TypeScript в поєднанні з NestJS як такої, що є однією з найбільш ефективних для використання в рамках вирішення поставлених задач; аналіз методів доставки застосунків на віддалені сервери, або ж розгортання застосунків; виокремлення найбільш оптимального для роботи з TypeScript.

Було виокремлено наступні завдання дослідження:

- Проаналізувати технології, що використовуються для back-end розробки та виокремити ті, які є найбільш ефективними для використаного в роботі прикладу застосунку.
- Обґрунтування переваг NestJS-стеку як серед JS/TS інструментів, так і інструментів на інших мовах програмування.
- Обґрунтування вибору та опис супутніх технологій, що були використані у роботі.

- Аналіз та виокремлення найбільш ефективних інструментів для розгортання застосунків, що використовують TypeScript.

Розділ 1. Дослідження та аналіз предметної області

1.1. Дослідження області back-end розробки

Back-end розробка – це набір апаратно-програмних засобів, який забезпечує логіку роботи веб-застосунку. Таке визначення було актуальним в епоху зародження Інтернету, коли функціонал клієнтської сторони веб-застосунків був значно обмеженішим ніж зараз, а про PWA(Progressive Web Application) не було і згадок. В сучасному ж світі, коли значною частиною логіки керує клієнтська сторона веб-застосунку, то вимоги і визначення того, як виглядає back-end застосунок суттєво змінилися.

Критерії, яким має відповідати сучасний back-end застосунок:

- Безпечність та захищеність
- Стійкість до високого навантаження
- Швидкодія
- Зрозумілий API

Також, кодова база сучасного back-end застосунку повинна бути гнучкою до змін, а отже структура має передбачати можливість збільшення обсягів коду, яке не призведе до того, що внесення правок буде ускладненим.

Кількість інженерів, що працюють з певною технологією завжди важлива під час її вибору, адже у випадку коли та чи інша технологія не є розповсюдженою, розробка та підтримка програмного продукту з її використанням ускладнюється.

У роботі розглядається розробка back-end застосунків на TypeScript, який є надмножиною JavaScript[3].

Проаналізуємо статистику з популярності мов програмування. За діаграмою розповсюдженості мов програмування, взятою з ресурсу

https://www.statista.com (див. додаток А), JavaScript є найпопулярнішою мовою програмування в світі, а TypeScript, який є його надмножиною знаходиться на 7 місці та ним володіє 30% респондентів[4].

Провівши дослідження відмінностей та особливостей найпопулярніших мов програмування, виокремимо критерії за якими для прикладу, розглянутого у роботі, TypeScript є більш доречним для використання:

- Python – значно поступається у швидкодії та має динамічну типізацію, на відміну від TypeScript, що значним чином впливає на швидкість внесення змін у кодову базу та на швидкість застосування рефакторингу.
- Java – потребує більш високого рівня абстракції, що знижує часто ключовий у розробці програмного забезпечення показник, Time-to-market та потребує більшої кількості налаштувань середовища для запуску застосунків та менеджменту залежностей.
- C/C++ – мають занадто низький рівень абстракції, що не дозволяє швидко та ефективно адаптувати застосунок до змін вимог до фінального продукту.

Порівняємо ефективність та здатність опрацьовувати велику кількість запитів різних високорівневих фреймворків[5].

Назва фреймворку	Кількість опрацьованих запитів на секунду
TypeScript - NestJS	67333
Java – Spring	23401
PHP – yii2	16006

Python – Django	15044
Ruby – Rails	8260

(наведені вище фреймворки були обрані за близькістю до NestJS в підходах до архітектури застосунку)

Як можна побачити з наведеної вище статистики, TypeScript в поєднанні з NestJS мають низку переваг над своїми конкурентами.

Порівнюючи з іншими популярними бібліотеками та фреймворками на NodeJS та TypeScript, можна відмітити, що рівень абстракції, який надають ці інструменти є досить низьким у порівнянні з NestJS, адже модульна структура фреймворку дозволяє написання програмного коду, з яким легко працювати навіть у великих проектах, а рефакторинг та внесення певних змін у поведінку застосунку можна проводити миттєво без побоювань щодо впливу на інші компоненти системи[6].

В сфері back-end розробки на NodeJS існує тенденція до переходу від більш низькорівневих Express.js та Fastify до NestJS, внутрішні механізми якого в свою чергу використовують Express.js або Fastify, тобто він є лише додатковим рівнем абстракції, або ж надбудовою над вищезгаданими бібліотеками.

1.2. Дослідження інструментів для розгортання back-end застосунків

Будь-який back-end застосунок зазвичай має два різні способи його запуску, або ж розгортання.

- Dev-build, або ж версія для розробки, яка запускається локально на пристроях розробників та потрібна для швидкої перевірки працездатності застосунку. Частіше за все передбачає hot reload.

- Production-build, або ж версія для запуску на сервері, що повинна бути надійною та займати якомога менше місця.

JavaScript, або ж TypeScript, який є його надмножиною – інтерпретована мова програмування, яка на відміну від деяких конкурентів, наприклад Golang, який компілюється у бінарний файл, що може бути запущений у будь-якому середовищі, потребує певного середовища запуску та менеджеру пакетів[7].

Зазвичай, для розгортання JavaScript застосунку на сервері використовують CLI-утиліту nvm, яка займається встановленням node.js з npm(менеджер пакетів для node.js) та менеджментом їхніх версій. Після завантаження потрібної версії node.js використовується npm для встановлення усіх залежностей проекту. І лише після того, як підготовка середовища для запуску застосунку завершена – використовується команда "node [filename]", де filename – це ім'я файлу, що потрібно запустити.

З NestJS останній крок відрізняється у тому, що він надає власний Command Line-інтерфейс для запуску застосунку у різних форматах, його dev- та production-версій.

Також бажано використовувати менеджер процесів, для управління різними застосунками, збору логів та підтримання їх запущеними поза рамками сесії терміналу, наприклад pm2, що додає складності до процесу розгортання node.js застосунку[8].

Отже, для запуску застосунку в такому форматі на вже існуючому сервері потрібно виконати наступні кроки:

- Завантажити nvm та pm2
- Встановити node.js з npm з допомогою nvm

- Завантажити вихідні коди застосунку
- Встановити усі залежності з допомогою npm
- Запустити застосунок з допомогою pm2

І цей процес потрібно буде частково пророблювати кожного разу після оновлення версії та додання правок в кодову базу.

Існує альтернативний варіант – використання Docker. Docker – це програмне забезпечення, яке дозволяє надати певному процесу обмежений доступ до ресурсів обчислювального пристрою(пам'ять, ОЗУ, мережа) та ізольовано запустити в рамках цього процесу необхідну операційну систему з рядом налаштувань[9].

Сукупність операційної системи та певних налаштувань застосованих до неї називається Docker Image, або ж образ. А процес запущений з образу, тобто програма з описаними відповідно до інструкції залежностями – Docker Container, або ж контейнер.

Для створення Docker-образу потрібно спочатку описати його у конфігураційному файлі з використанням спеціального набору команд, які підтримуються ядром системи контейнеризації, а потім виконати побудову образу з шарів, які створюються після виконання кожної інструкції конфігураційного файлу. Для створення контейнеру потрібно просто вказати системі, що називається Docker Engine, екземпляр якого образу потрібно відтворити.

Особливістю Docker та ключовою його перевагою над інструментом, який займав домінуючу позицію на цьому ринку – VM(Віртуальна Машина) є те, що він абсолютно не залежить від платформи і один образ Docker працюватиме однаково на Windows, MacOS та Linux. Це дає можливість розробляти програмні продукти в Docker-і, що гарантуватиме

однозначність виконання описаних інструкцій та попереджатиме випадки того, що якийсь код працюватиме на комп’ютері розробника, але не працюватиме на сервері, або ж навпаки[10].

З використанням інструментів, які надає Docker, запуск застосунку зводиться до повторного створення образу та перезапуску контейнеру вже з нового образу, а враховуючи, що Docker використовує багатошаровий підхід, то інструкції, що виконують завантаження залежностей, часто кешуються, тому створення нового образу займає досить короткий проміжок часу та невеликий обсяг обчислювальних потужностей. А перезапуск контейнеру зазвичай займає не набагато більше часу, ніж виконання операції знищення процесу ОС та операції створення нового процесу, адже саме ці дії виконує Docker Engine, вирішуючи задачу перезапуску контейнеру[11].

Отже, використання Docker має наступні переваги:

- дозволяє описати кроки створення образу, що веде до часткової або повної автоматизації процесу запуску застосунку на сервері
- дозволяє розробляти та розгортати застосунок у абсолютно однаковому середовищі, що зменшує кількість проблем у розробці та проблем пов’язаних з відтворенням поведінки на різних обчислювальних пристроях

Вищезазначені переваги роблять Docker універсальним інструментом, який значно спрощує розробку та доставку програмного забезпечення, через що існує значна тенденція до відмови від традиційного варіанту розгортання застосунків та переходу до контейнеризації.

Отже, можемо однозначно ствердити, що використання Docker виправдане та доцільне у контексті розробки back-end застосунків з використанням TypeScript.

Розділ 2. Проектування та розробка системи

2.1. Опис складових системи та використаних для розробки технологій

Прикладом системи було обрано застосунок для ведення складського обліку, адже такий вид програмних систем досить ефективно розробляти з використанням описаного вище фреймворку. Застосунки такого типу потребують створення великої кількості ендпоінтів для виконання різноманітних операцій, а NestJS дозволяє ефективне написання цих ендпоінтів, при цьому підтримуючи високий рівень абстракції завдяки модульному підходу до проектування структури проекту[12].

Наступною невід’ємною частиною будь-якого back-end застосунку є взаємодія з базою даних. Було прийнято рішення використовувати PostgreSQL як СКБД в рамках цієї роботи. Рішення виправдане тим, що для даних, які опрацьовуються та зберігаються в рамках цього проекту зручно використовувати реляційну базу даних, серед яких PostgreSQL є лідером, адже опереджає своїх конкурентів за швидкодією та набором наявного функціоналу. Для взаємодії з базою даних було вирішено використовувати ORM. ORM(Object-relational mapping, об’єктно-реляційне відображення) – це техніка програмування, суть якої полягає в тому, що дані, які використовуються в моделях/сутностях в програмному коді та поля бази даних можуть бути зв’язані, створюючи так звану “вітруальну об’єктну базу даних”. Тобто сутності з текстів програми можуть бути конвертовані в таблиці, а екземпляри цих сутностей в рядки таблиці, і навпаки від бази даних до об’єктів[13]. Таких підхід має ряд переваг над традиційним підходом до роботи з базами даних через SQL-запити:

- Автоматизація багатьох процесів під час розробки
- Зменшення об’ємів коду
- Прозоре кешування об’єктів на рівні застосунку

- Підвищення швидкості обслуговування додатку та внесення змін

Серед запропонованих ORM для TypeScript з відкритим кодом, що ефективні і поєднанні з NestJS, найкращим варіантом є TypeORM[14]. У NestJS є відповідні модулі, що дозволяють легку інтеграцію сутностей TypeORM в код на NestJS, тому цей варіант є найзручнішим та найпоширенішим для роботи з базами даних з back-end застосунку на NestJS.

Окрім основного фреймворку, що було використано в процесі розробки та ORM, до залежностей проекту входять:

- bcrypt – для хешування паролів користувачів та перевірки співпадіння хешу
- class-validator – для валідації запитів до API
- dotenv – для завантаження конфігурації з .env файлу або змінних оточення
- passport – для роботи з аутентифікацією та авторизацією

2.2. Налаштування контейнеризації та мережевої взаємодії

Як зазначено раніше, розробка та розгортання застосунку, розглянутого в рамках цієї роботи, проводитиметься з використанням інструментів, що надає Docker.

В ході проектування застосунку було прийняте рішення, що на поточному етапі розробки, його буде розділено на дві ключові частини. 1 – сервіс для управління складським обліком, 2 – сервіс для запису та обслуговування користувацьких облікових записів. Наступним питанням був вибір архітектури застосунку між монолітною та мікросервісною. Монолітна

архітектура дозволяє зекономити час та зусилля при розробці невеликого застосунку, адже відкидаються проблеми мережевої взаємодії та підтримки декількох паралельно працюючих застосунків, з іншого боку, мікросервісна архітектура дозволяє бути більш гнучкими в майбутньому з точки зору інфраструктурних рішень та відкриває багато можливостей розвитку системи з технічної сторони в майбутньому[15]. З огляду на те, що у проекту планується подальший розвиток і розробка комплексної системи на базі перших сервісів, було прийнято рішення обрати мікросервісний підхід, тим паче використання Docker суттєво спрощує цей процес, надаючи механізми для легкого налаштування міжсервісної взаємодії та оптимізації процесів розгортання мікросервісів.

Для кожного мікросервісу було розроблено два Dockerfile – конфігураційні файли Docker: один для створення dev-образу для розробки, інший – для створення production-образу для розгортання застосунку на сервері(див. додаток Б). Обидва файли використовують образ “node:18” як базовий. Цей образ підтримується OpenJS Foundation та складається з linux-середовища з встановленим Node.js 18 версії та npm. Використання такого образу дозволяє замінити усі кроки по налаштуванню середовища з традиційного підходу до розгортання застосунків.

Розглянемо особливості кожного з Dockerfile-ів:

- Dockerfile.prod містить повний набір інструкцій для встановлення усіх залежностей проекту та його запуску, але при значній кількості залежностей розмір результуючого образу може бути настільки великим, що його завантаження на сервері займатиме занадто багато часу, а контейнер з цим образом багато пам’яті на пристрої. Для цього було використано multi-stage-build, або ж багатоетапну побудову образу, використовуючи “node:18” та “node:18-alpine” як базові

образи. Alpine – дистрибутив Linux, який орієнтований на безпеку, невибагливість до ресурсів та мінімізацію розміру операційної системи. Отже, Alpine Linux використовується в другому етапі побудови образу з метою мінімізації розміру результуючого образу, адже окрім того, що це дозволило зменшити вагу базового образу, частину залежностей, які використовуються лише для розробки було відкинуто командою “RUN npm prune --production”, тому фінальний docker image буде значно легшим.

- Dockerfile.dev містить лише декілька інструкцій, що описують базовий образ та запускають проект, але не містить копіювання вихідних текстів програми. Зроблено це задля того, щоб використати docker volumes, або томи. Томи – альтернативний підхід до перенесення даних всередину контейнеру, який передбачає монтування певної директорії з файлової системи хоста в файлову систему контейнеру. Цей підхід дозволяє уникнути копіювання файлів та дозволяє використання однієї ділянки файлової системи одночасно хостом та контейнером, що необхідно для розробки програмного забезпечення з використанням Docker, адже використовуючи цей підхід, при кожному запуску контейнера не відбувається завантаження залежностей, при цьому редагування коду призводить до його редагування в контейнері та викликає hot-reload. Також, це дозволяє розробляти застосунок з використанням тієї самої операційної системи, що буде використана для його запуску на сервері, тому неочевидну поведінку, що специфічна лише для певних ОС можна попередити на етапі розробки.

Оскільки система складається з двох мікросервісів та СКБД, які повинні бути об’єднані однією мережею, було розглянуто опції для налаштування мережевої взаємодії. Коли docker engine запускає контейнер, він

автоматично додається до стандартної мережі, яка називається bridge network, кожному контейнеру присвоюється власна ip-адреса, але вони не можуть взаємодіяти між собою, адже не знають адреси один одного. В класичній мікросервісній архітектурі це питання вирішується завдяки service-discovery. Розглянемо на прикладі Consul: існує сервіс, який отримує від кожного мікросервісу його ip-адресу і псевдонім, який буде використано в подальшому; коли якийсь мікросервіс звертатиметься до іншого за псевдонімом – Consul перенаправить його на потрібну ip-адресу. Але таке рішення потребує значної кількості налаштувань, не дуже зручне для розробки, адже вимагає суттєвих зусиль для щоразової перевірки працездатності застосунку та не є виправданим для обсягів системи, використаної як приклад в цій роботі. Тому в роботі було використано docker compose. Docker compose – це інструмент для запуску застосунків, що складаються з багатьох контейнерів та потребують ізольованого середовища для їхньої комунікації. Було розроблено docker-compose файл для одночасного запуску усіх контейнерів з можливістю вибору dev або production конфігурацій(див. додаток В).

У docker-compose файлі було створено загальну мережу та додано псевдоніми для контейнерів всередині мережі, тому незалежно від типу запуску(dev/production), сервіси зможуть звертатися один до одного за псевдонімами замість ip-адрес, а мережі Docker та dns-резолвери замінять псевдонім на адресу шуканого контейнеру.

2.3. Розробка API

NestJS використовує модульний підхід до організації компонентів системи. Загалом, кожен модуль має чотири опційних поля, що описують його поведінку та взаємодію з іншими модулями:

- imports – список модулів, які імпортуються в поточний
- controllers – список контролерів, які були визначені в цьому модулі та повинні бути проініціалізовані
- providers – список сервісів, що надають певні дані
- exports – список підмножини сервісів, що надаються цим модулем та повинні бути доступні в інших модулях, які імпортують поточний

В корені кожного мікросервісу існує два файли: *main.ts* та *app.module.ts*(див. Додаток Г). Перший є просто вхідною точкою, в якій з використанням патерну фабрики з кореневого модулю AppModule та додаткових конфігураційних опцій, створюється об'єкт застосунку, в якого викликається метод “listen” для запуску веб-серверу на порті 3000. Другий – файл з кореневим модулем(створений з використанням декоратору “@Module”), описує те, які модулі проекту повинні бути ініціалізовані з запуском проекту, та як можна помітити, не містить жодного контролера, адже загальна суть використання кореневого модуля NestJS – імпорт усіх дочірніх модулів, які вже в свою чергу містять в собі контролери.

Контролери у використовуваному фреймворку описуються завдяки декоратору “@Controller”, який приймає у аргументи параметр “prefix” та помічає клас контролером, що може відповідати на вхідні HTTP запити(requests) продукуючи HTTP відповіді(responses). Такий контролер визначає контекст для одного або більше обробників, що викликаються коли шлях та HTTP метод запиту співпадають з визначеним для конкретного обробника. Обробники визначаються як функції в рамках класу-контролера, а HTTP метод та шлях визначаються одним з декораторів зі списку: [“@Get”, “@Delete”, “@Post”, “@Patch”, “@Put”, “@Options”, “@Head”, “@All”]. Приклад реалізації контролеру для створення та редагування користувача(див. Додаток Д). Як можна побачити в додатку,

змінну для тіла HTTP запиту можна зв'язати з ним за допомогою декоратора "@Body" та об'явити її типом DTO(Data Transfer Object), в поля якого будуть записані дані з тіла запиту. NestJS пропагує підхід з DTO до отримання тіла запиту, адже такий підхід дозволяє легко передавати одну сутність між компонентами системи та проводити додаткові маніпуляції над нею. Однією з таких маніпуляцій є валідація інформації, що знаходиться в тілі запиту. В звичайному підході, наприклад з використанням словників для зберігання тіла запиту, валідацію запиту потрібно писати в явному вигляді, перевіряючи значення кожного елемента словника. В випадку з DTO можна використовувати додаткові інструменти для лаконічного запису обмежень до певних даних. Наприклад модуль "class-validator" дозволяє використовувати декоратори, що розміщаються перед кожним полем класу DTO та встановлюють певні обмеження до значень, яких воно може набувати. Приклад описання DTO та валідації його полів(див. Додаток Е). Підхід з використанням досить зручний, адже можна використовувати заготовлені декоратори як "@IsEmail", "@IsString", "@MinLength", так і визначати власні, які зручні для використання в рамках проекту.

Також, з додатку Д можна побачити, як один з провайдерів, в цьому випадку типу "UserService" був ініціалізований в конструкторі. Вихідний код UserService(див Додаток. Є). Декоратор "@Injectable" застосовано до класу UserService, що означає, що його було помічено як Provider, а отже він може бути введений в інші класи з використанням параметрів конструктору, що використовують вбудовану в NestJS систему ін'єкції залежностей(Dependency Injction)[16]. Також, в конструкторі цього сервісу була проведена ін'єкція репозиторію для доступу до об'єктів бази даних, з використанням узагальнення та сутності User для конкретизації узагальнення. Всередині класу UserService з використанням асинхронного

синтаксису TypeScript та введеного репозиторію, описані основні методи, які дозволяють операції створення та редагування користувача. Ці методи можуть бути використані в усіх класах(переважно контролерах), що використовують провайдер UserService. Наявність таких інструментів побудови модульності з використанням ін'єкції залежностей робить NestJS досить складним для освоєння фреймворком, але значно покращує структурованість системи та підвищує ефективність розробки.

Висновки

Було досліджено різноманітні інструменти, що покривають повний цикл back-end розробки, проаналізовані їхні переваги та недоліки, розглянуто найоптимальніші підходи для вирішення поставленої задачі. У роботі використано актуальні та популярні на час написання технології: TypeScript, NestJS з TypeORM, PostgreSQL та Docker. На прикладі застосунку для ведення складського обліку було продемонстровано зручність та доречність вибору технологій. У майбутніх дослідженнях може розглядатися трансформація застосунку в cloud-native з введенням додаткових інструментів та розгортанням на базі одного з провайдерів хмарних сервісів.

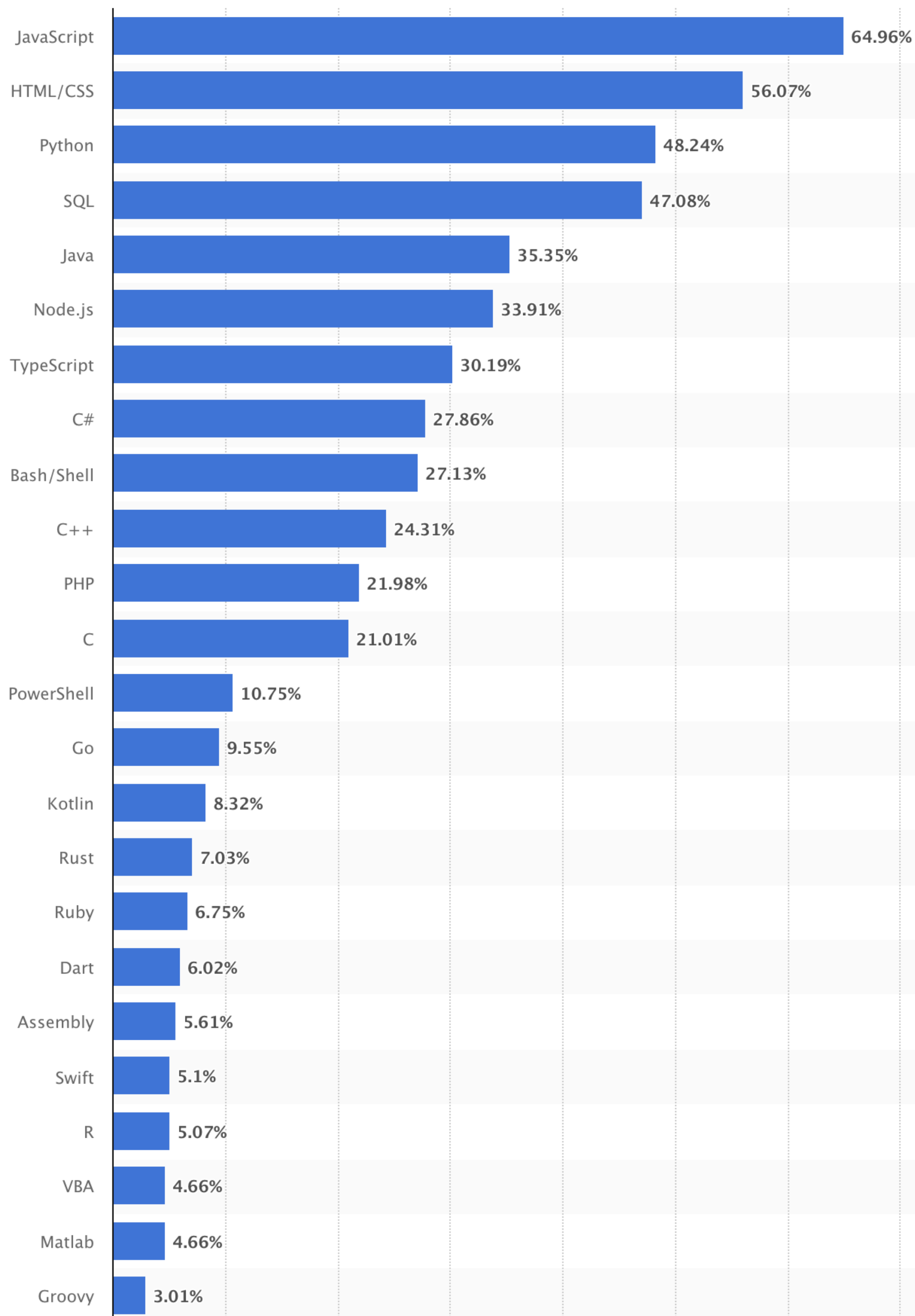
Список використаних джерел

1. How Many Software Developers Are in the US and the World? [Електронний ресурс] // DAXX by Grid Dynamics. – 2021. – Режим доступу до ресурсу: https://www.daxx.com/blog/development-trends/number-software-developers-world.
2. Determining Software Time-to-Market and Testing Stop Time when Release Time is a Change-Point [Електронний ресурс]// International Journal of Mathematical, Engineering and Management Sciences – 2019 / Singh, Ompal; Panwar, Saurabh; Kapur, P K. – Режим доступу до ресурсу: https://www.proquest.com/docview/2632622215
3. Understanding TypeScript / Gavin Bierman, Martín Abadi, 2014
4. A large scale study of programming languages and code quality in github [Електронний ресурс] / Baishakhi Ray, Daryl Posnett, Vladimir Filkov – 2014 – Режим доступу до ресурсу: https://dl.acm.org/doi/abs/10.1145/2635868.2635922
5. Web Framework Benchmark [Електронний ресурс] // TechEmpower – 2021 – Режим доступу до ресурсу: https://www.techempower.com/benchmarks/
6. Developing a back-end of a web application with NestJS framework / Anh Duc Pham – 2020 – Режим доступу до ресурсу: https://www.theseus.fi/bitstream/handle/10024/353200/Pham_Duc.pdf?sequence=2
7. The MEAN Stack [Електронний ресурс] // IRJET Journal – 2017 – Режим доступу до ресурсу: https://www.academia.edu/download/53592508/IRJET-V4I5795.pdf
8. Advanced, production process manager for Node.js: документація [Електронний ресурс]. Режим доступу до ресурсу: https://pm2.keymetrics.io
9. Docker: документація [Електронний ресурс]. Режим доступу до ресурсу: https://docs.docker.com
10. Docker: Lightweight Linux Containers for Consistent Development and Deployment [Електронний ресурс] / Dirk Merkel – Режим доступу до ресурсу: https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf

11. Optimize Windows Dockerfiles [Електронний ресурс] // Microsoft – 2022 – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-docker/optimize-windows-dockerfile
12. NestJS, a progressive Node.js framework: документація [Електронний ресурс]. Режим доступу до ресурсу: https://nestjs.com/
13. Natural Language Interface to Relational Database (NLI-RDB) Through Object Relational Mapping (ORM) / Abdullah Alghamdi, Majdi Owda & Keeley Crockett – 2016 – Режим доступу до ресурсу: https://link.springer.com/chapter/10.1007/978-3-319-46562-3_29
14. TypeORM – Amazing ORM for TypeScript and JavaScript: документація [Електронний ресурс]. Режим доступу до ресурсу: https://typeorm.io/
15. Monolith to Microservices / Sam Newman // O'REILLY – 2019 – Режим доступу до ресурсу: https://books.google.se/books?id=ota_DwAAQBAJ
16. Dependency Injection: Design patterns using Spring and Guice / Dhanji R. Prasanna – 2009 – Режим доступу до ресурсу: https://books.google.se/books?hl=ru&lr=&id=PzkzEAAAQBAJ

Додатки

Додаток А



Додаток Б

Dockerfile для розгортання та розробки відповідно

Dockerfile.prod

```
FROM node:18 AS builder

ARG NODE_ENV=production
ENV NODE_ENV=${NODE_ENV}

WORKDIR /app

COPY users/ .

RUN npm i
RUN npm run build
RUN npm prune --production

FROM node:18-alpine AS production

COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/package-lock.json ./package-lock.json
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules

CMD ["sh", "-c", "npm run start:prod"]
```

Dockerfile.dev

```
FROM node:18

WORKDIR /app

CMD ["npm", "run", "dev"]
```

Додаток В

docker-compose.yml файл для запуску багатьох контейнерів в одній мережі

```yaml
version: "3.9"
services:
  users-dev:
    container_name: users
    volumes:
      - ./users:/app
    build:
      dockerfile: users/Dockerfile.dev
    ports:
      - "20001:3000"
    env_file:
      - users/.env
    networks:
      - selso-network
    profiles:
      - dev

  users-prod:
    container_name: users
    build:
      dockerfile: users/Dockerfile
    ports:
      - "20001:3000"
    env_file:
      - users/.env
    networks:
      - selso-network
    profiles:
      - prod

  warehouse-dev:
    container_name: warehouse
    volumes:
      - ./warehouse:/app
    build:
      dockerfile: warehouse/Dockerfile.dev
    ports:
      - "20002:3000"
    networks:
      - selso-network
    profiles:
      - dev

  warehouse-prod:
    container_name: warehouse
    build:
      dockerfile: warehouse/Dockerfile
    ports:
      - "20002:3000"
    networks:
      - selso-network
    profiles:
      - prod

  postgres:
```

```yaml
    container_name: selso-db
    image: postgres:14
    volumes:
      - ./var/pgdata:/var/lib/postgresql/data
      - ./docker/postgres/init.sql:/docker-entrypoint-initdb.d/init.sql
    environment:
      POSTGRES_USER: "${POSTGRES_USER}"
      POSTGRES_PASSWORD: "${POSTGRES_PASSWORD}"
    ports:
      - "21001:5432"
    networks:
      - selso-network
    profiles:
      - dev
      - prod


networks:
  selso-network:
    driver: bridge
```

Додаток Г

Код файлів кореневої директорії

main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    logger: ['warn', 'error'],
  });
  app.useGlobalPipes(
    new ValidationPipe({
      disableErrorMessages: false,
    }),
  );
  await app.listen(3000);
}

bootstrap();
```

app.module.ts

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { configService } from './config/config.service';
import { UserModule } from './user/user.module';
import { AuthModule } from './auth/auth.module';
import { JwtAuthGuard } from './auth/jwt-auth.guard';

@Module({
  imports: [
    TypeOrmModule.forRoot(configService.getTypeOrmConfig()),
    UserModule,
    AuthModule,
  ],
  providers: [{ provide: 'APP_GUARD', useClass: JwtAuthGuard }],
})
export class AppModule {}
```

Додаток Д

```
import { BadRequestException, Body, Controller, Patch, Post, Req } from
'@nestjs/common';
import { UserService } from './user.service';
import { UserDto } from './dto/user.dto';
import { UpdateUserDto } from "./dto/updateUser.dto";

@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Public()
  @Post()
  async createUser(@Body() dto: UserDto) {
    const resp = await this.userService.create(dto);
    return { id: resp.id };
  }

  @Patch()
  async updateUser(@Body() dto: UpdateUserDto, @Req() req: any) {
    return await this.userService.update(dto);
  }
}
```

Додаток Е

```
import { IsEmail, MinLength, IsString, Matches, IsOptional, IsBoolean, 
IsNumber } from 'class-validator';

export class UserDto {
  @IsEmail()
  email: string;

  @IsString()
  @MinLength(8, {message: 'password must be at least 8 characters long'})
  @Matches(/((?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$/, 
{message: 'password too weak'})
  password: string;

  @IsString()
  firstName: string;

  @IsOptional()
  @IsString()
  middleName: string | null;

  @IsString()
  lastName: string;

  birthDate: Date;

  @IsString()
  phone: string;

  @IsOptional()
  @IsBoolean()
  subscribedToNewsletter: boolean | null;

  @IsOptional()
  @IsNumber()
  permissionLevel: number | null;
}
```

Додаток Є

```
import { User } from './user.entity';
import { UserDto } from './dto/user.dto';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { hash, compare } from 'bcrypt';
import { UpdateUserDto } from './dto/updateUser.dto';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User) private readonly repo: Repository<User>,
  ) {}

  async create(user: UserDto): Promise<User> {
    const hashedPassword = await hash(user.password, 10);
    return await this.repo.save({ ...user, password: hashedPassword
}).catch();
  }

  async authenticate(email: string, password: string): Promise<User> {
    const user = await this.repo.findOne({ email: email });

    if (!user) throw new UnauthorizedException('User not found');

    const passwordsMatch = await compare(password, user.password);
    if (!passwordsMatch) {
      throw new UnauthorizedException("Passwords don't match");
    }

    return user;
  }

  async update(user: UpdateUserDto): Promise<User> {
    return this.repo.save(user);
  }
}
```